deep se
dependable evolvable pervasive software engineering group

Smart Pointers

Federica Filippini - Marco Lattuada - Danilo Ardagna

Politecnico di Milano
federica.filippini@polimi.it
marco.lattuada@polimi.it
danilo.ardagna@polimi.it

POLITECNICO DI MILANO

Content

- Pointers
- What is a smart pointer?
- C++ 11 smart pointers
  - `std::shared_ptr`

Memory management in C++

- Most of the programs we've written so far have used objects that have well-defined lifetimes } we always checked scopes
- C++ lets us allocate objects dynamically
  - Dynamically allocated objects have a lifetime that is independent of where they are created; they exist until they are explicitly freed
- Programs use the **free store** or **heap** for objects that they **dynamically allocate** (i.e., at run time)
  - The program controls the lifetime of dynamic objects
  - Code must explicitly destroy such objects when they are no longer needed !

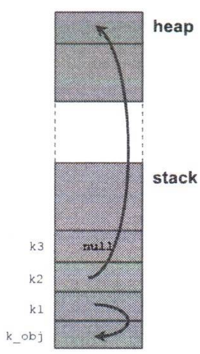
Pointers in action

```
Kitten k_obj;

Kitten* k1 = &k_obj;

Kitten* k2 = new Kitten;

Kitten* k3 = nullptr;
```

k2 points to a piece of dynamically allocated memory

this will point to nothing until we give him a value

because of this, we'll have to delete (free) explicitely (the pointer must be delated explicitely)



Memory management in C++

- Properly freeing dynamic objects turns out to be a surprisingly rich source of bugs
- Biggest question is how to ensure that allocated memory will be freed when it is no longer in use
  - If we forget to free the memory we have a memory leak
  - If we free the memory when there are still pointers referring to that memory, we have a pointer that refers to memory that is no longer valid (dangling pointer)
  - If we subsequently delete the other pointers, then the free store may be corrupted
- These kinds of errors are **considerably easier to make than** they are to **find** and **fix!!!**

Memory leakage

```
double* calc(int result_size, int max)
{
    double* p = new double[max]; // allocate another max
                                 // doubles
                                 // i.e., get max doubles from
                                 // the free store
    double* result = new double[result_size];
    // … use p to calculate results to be put in result …
    return result;
}

double* r = calc(200,100); // oops! We "forgot" to give the memory
                           // allocated for p back to the free
                           // store
```

new $\xrightarrow{!}$ delete

Memory leakage

```
double* calc(int result_size, int max)
{
    double* p = new double[max];
    double* result = new double[result_size];

    // … use p to calculate results to be put in result …
    delete[] p;
    return result;
}

double* r = calc(200,100);
delete[] r;                       // easy to forget
```


Memory leakage (another example)

```
// factory returns a pointer to a dynamically allocated object
// the caller must remember to delete the memory
Foo* factory(T arg)
{
    // process arg as appropriate
    return new Foo(arg); // caller is responsible for deleting
                         // this memory
}

void use_factory(T arg)
{
    Foo *p = factory(arg); // use p but do not delete it
} // p goes out of scope, but the memory to which p points is not freed!
```


Memory leakage (another example)

```
// factory returns a pointer to a dynamically allocated object
// the caller must remember to delete the memory
Foo* factory(T arg)
{
    // process arg as appropriate
    return new Foo(arg); // caller is responsible for deleting
                         // this memory
}

void use_factory(T arg)
{
    Foo *p = factory(arg);
    delete p;
}
```

however to know that we have to add a delete we also have to know that a "new" have been used. If we're working only on use-factory() we may not know it.

Smart pointers

- To make using dynamic objects safer, the library defines smart pointer types that manage dynamically allocated objects
- Smart pointers ensure that the objects to which they point are automatically freed when it is appropriate to do so
- Goal: implement pointer-like objects in simple and leak-free programs

Smart pointers evolution

- `boost::scoped_ptr` has problems with a C-style array!
- `std::auto_ptr` deprecated!
- `std::unique_ptr`
- `std::weak_ptr`
- **`std::shared_ptr`**

C++ 11

we'll use only this

C++ 11 smart pointers

- **shared_ptr**
  - allows multiple pointers to refer to the same object
- **unique_ptr**
  - "owns" the object to which it points → advanced feature (APSC)

- Both are defined in the memory header
- Implemented through Templates

which means that we can have a shared pointer to double, integer, string, ..

What is a "smart pointer?"

- Loose definition: object that behaves like a pointer, but somehow "smarter"

a smart pointer deletes memory when it is appropriate to do so: it detects when to delete memory by itself

- Major similarities to raw pointers:
  - Is bound to 0 or 1 objects at a time, often can be re-bound
  - Supports indirection: operator *, operator ->

- Major differences:
  - Has some "smart feature"
    - Automatic deletion of the owned object
  - Iterators: **it++**

C++ 11 Shared Pointers

#include<memory>

The shared_ptr Class

```
shared_ptr<string> p1;   // shared_ptr that can point at a string
shared_ptr<list<int>> p2;  // shared_ptr that can point at a list of ints
```

shared_ptr<type>

```
// if p1 is not null, check whether it's the empty string
if (p1 && p1->empty())
      *p1 = "hi";  // if so, dereference p1 to assign a new value
                   // to that string
```

if(p1) means if(p1 points to an object)

the whole condition is: if p1 points to an object and this object is the null string. The order is important since we cannot access to the object pointed if the object doesn't exist

(it's an **error** if we access a ∄ object)

The make_shared Function

- Safest way to allocate and use dynamic memory
  - Allocates and initializes an object in dynamic memory and returns a shared_ptr that points to that object

```
// shared_ptr that points to an int with value 42
shared_ptr<int> p3 = make_shared<int>(42);

// p4 points to a string with value 9999999999
shared_ptr<string> p4 = make_shared<string>(10, '9');

// p5 points to an int that is value initialized to 0
shared_ptr<int> p5 = make_shared<int>();

// p6 points to a dynamically allocated, empty vector<string>
auto p6 = make_shared<vector<string>>();

auto p = make_shared<int>(42); // object to which p points has one user
auto q(p); // p and q point to the same object
           // object to which p and q point has two users
```

!

!

!

!!

The best ways to initialize shared pointers are "make-shared" and the copy initialization

shared_ptr implementation

How can this pointer be "smart"?

- We can think of a `shared_ptr` as if it has an associated counter, usually referred to as a **reference count**
- Whenever we copy a `shared_ptr`, the count is incremented
- The counter is decremented when we assign a new value to the `shared_ptr` and when the `shared_ptr` itself is destroyed (e.g., when a local `shared_ptr` goes out of scope)
- **Once a `shared_ptr` counter goes to zero, the `shared_ptr` automatically frees the object that it manages**

This avoids memory leaks!

shared_ptr operations

shared_ptr<T> sp	Null smart pointer that can point to objects of type T
p	Use *p* as a condition; true if *p* points to an object
*p	Dereference *p* to get the object to which *p* points
p->mem	Same as *(*p).mem*
p.get()	Returns the pointer in *p*. **Be very careful!**
swap(p,q) p.swap(q)	Swap the pointers in *p* and *q*

If we initialize a pointer without "make_shared" or without copy (or else) the pointer will be initialized with the NULL POINTER

! this returns the row pointer that is embedded in p (we can see a shared pointer as a row pointer + a counter, if we use the ".get()" we get the row pointer embedded) However, this should not be done (we would still have the same problems)

shared_ptr operations

make_shared<T> *args*	Returns a `shared_ptr` pointing to a dynamically allocated object of type T; use *args* to initialize that object
shared_ptr<T> p(q)	*p* is a copy of the shared_ptr *q*; increments the count in q. The pointer in *q* must be convertible to T*
p = q	*p* and *q* are `shared_ptrs` holding pointers that can be converted to one another. Decrements *p* reference count and increments *q* count; deletes *p* existing memory if *p* count goes to 0
p.unique()	Returns true if *p* count is one; false otherwise
p.use_count()	Returns the number of objects sharing with *p*; slow, use for debugging

this operation increments the counter of the copied pointer

we assign to p the value of q. if p already pointed to an object, this operation decrements its counter

= true if p is the unique pointer pointing to an object

returns the value of the p counter (= the number of pointers that point to *p)

shared_ptr implementation

shared_ptrs automatically destroy their objects ...
...and automatically free the associated memory

```
// factory returns a shared_ptr pointing to a dynamically allocated object
shared_ptr<Foo> factory(T arg)
{
    // process arg as appropriate
    // shared_ptr will take care of deleting this memory
    return make_shared<Foo>(arg);
}

void use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg); // use p
} // p goes out of scope; the memory to which p points is automatically freed
```

shared_ptr implementation

shared_ptrs automatically destroy their objects ...
...and automatically free the associated memory

```
// factory returns a shared_ptr pointing to a dynamically allocated object
shared_ptr<Foo> factory(T arg)
{
    // process arg as appropriate
    // shared_ptr will take care of deleting this memory
    return make_shared<Foo>(arg);
}
shared_ptr<Foo> use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);// use p
    return p; // reference count is incremented when we return p
} // p goes out of scope; the memory to which p points is not freed
```

!!!

because until this return the counter of p was 1, with the return the counter has +1, going out of scope is -1

Classes with resources that have Dynamic Lifetime

- Programs use dynamic memory for one of three purposes:
  1. They don't know how many objects they'll need
  2. They don't know the precise type of the objects they need
  3. They want to share data between several objects

```
vector<string> v1; // empty vector
{   // new scope
    vector<string> v2 = {"a", "an", "the"};
    v1 = v2; // copies the elements from v2 into v1
} // v2 is destroyed, which destroys the elements in v2
// v1 has three elements, which are copies of the ones originally in v2
```


Classes with resources that have Dynamic Lifetime

- Some classes allocate resources with a lifetime that is independent of the original object
- Assume we want to define a class *LPVector* that will hold a collection of elements
- Unlike the vector, we want that LPVector objects which are copies of one another to share the same elements (like-a-pointer)
- In the following we will consider a LPVector specialization to store string: *StrLPVector*

DEMO

Classes with resources that have Dynamic Lifetime

- In general, when two objects share the same underlying data, we can't unilaterally destroy the data when an object of that type goes away

```
StrLPVector b1; // empty StrLPVector
{ // new scope
    StrLPVector b2 = {"a", "an", "the"};
    b1 = b2; // b1 and b2 share the same elements
} // b2 is destroyed, but the elements in b2 must not be destroyed
// b1 points to the elements originally created in b2
```

to obtain this behaviour

Defining the StrLPVector Class

- We **can't store** the **vector directly** in a **StrLPVector object**
  - Members of an object are destroyed when the object itself is destroyed
  - If b1 and b2 are two StrLPVector that share the same vector. If that vector were stored in one of those StrLPVector —say, b2—then that vector, and therefore its elements, would no longer exist once b2 goes out of scope
- To ensure that the elements continue to exist, we'll store the **vector** in **dynamic memory**
- We'll give each StrLPVector a shared_ptr to a dynamically allocated vector
  - That shared_ptr member will keep track of how many StrLPVector share the same vector and will delete the vector when the last StrLPVector using that vector is destroyed

Defining the StrLPVector Class

```
class StrLPVector {
public:
    typedef std::vector<std::string>::size_type size_type;

    StrLPVector();
    StrLPVector(std::initializer_list<std::string> il);

    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }

    // add and remove elements
    void push_back(const std::string &t) {data->push_back(t);}
    void pop_back() {data->pop_back();};

    // element access
    std::string& front() {return data->front();}
    std::string& back() {return data->back();}

private:
    std::shared_ptr<std::vector<std::string>> data;
    // write msg if data[i] isn't valid
};
```

we're adding a dynamical part

StrLPVector Constructors

```
StrLPVector::StrLPVector():
data(make_shared<vector<string>>()) { }

StrLPVector::StrLPVector(initializer_list<string> il):
    data(make_shared<vector<string>>(il)) { }
```


Copying, assigning, and destroying StrLPVectors

- StrLPVector uses the default versions of the operations that copy, assign, and destroy objects of its type
  - These operations copy, assign, and destroy the data members of the class (in this case only its `shared_ptr`)

Copying, assigning, and destroying StrLPVectors

- When we copy, assign, or destroy a StrLPVector, its shared_ptr member will be copied, assigned, or destroyed
  - Copying a shared_ptr increments its reference count
  - Assigning one shared_ptr to another increments the count of the right-hand operand and decrements the count in the left-hand operand
  - Destroying a shared_ptr decrements the count
- If the count in a shared_ptr goes to zero, the object to which that shared_ptr points is automatically destroyed
  - The vector allocated by the StrLPVector constructors will be automatically destroyed when the last StrLPVector pointing to that vector is destroyed

Be wary of shared ownership

- Do not design your code to use shared ownership without a very good reason
  - One such reason is to avoid expensive copy operations, but you should only do this if the performance benefits are significant, and the underlying object is immutable (i.e. shared_ptr<const Foo>)
- In many cases copies can be avoided by correctly using references
- If you do use shared ownership, prefer to use `shared_ptr`

shared_ptr and built-in pointers

- The smart pointer constructors that take pointers are explicit
  - We cannot implicitly convert a built-in pointer to a smart pointer; we must use the direct form of initialization to initialize a smart pointer

this creates a dynamically allocated object and a row pointer that points it. Then the row pointer is converted into a shared pointer explicitly. This cannot be done.

```
shared_ptr<int> p1 = new int(1024); // error: must
                                    // use direct
                                    // initialization
shared_ptr<int> p2(new int(1024)); // ok: uses
                                   // direct
                                   // initialization
```

- The initialization of p1 implicitly asks the compiler to create a shared_ptr from the int* returned by new. Because we can't implicitly convert a pointer to a smart pointer, this initialization is an error

shared_ptr and built-in pointers

- For the same reason, a function that returns a shared_ptr cannot implicitly convert a plain pointer in its return statement:

```
shared_ptr<int> clone(int p) {
    // error: implicit conversion to shared_ptr<int>
    return new int(p);
}
```

- We must explicitly bind a shared_ptr to the pointer we want to return:

```
shared_ptr<int> clone(int p) {
    // ok: explicitly create a shared_ptr<int> from int*
    return shared_ptr<int>(new int(p));
}
```


Don't Mix Ordinary Pointers and Smart Pointers!

- A `shared_ptr` can coordinate destruction only with other `shared_ptrs` that are copies of itself
  - This fact is one of the reasons it is recommend using `make_shared` rather than `new`
  - We bind a `shared_ptr` to the object at the same time that we allocate it
  - There is no way to inadvertently bind the same memory to more than one independently created `shared_ptr`
- It is dangerous to use a built-in pointer to access an object owned by a smart pointer, because we may not know when that object is destroyed

!!

Which type of pointer should I use?

Raw Pointer

- When you need to store **addresses** of **existing variables**

we cannot use a shared pointer to store the address of an existing variable. Every time we initialize a smart pointer we create dynamically a new variable.

```
int a = 10;
int* ptr_a = &a;
*ptr_a = 15;
```

Smart pointer

- When you want to declare a **new dynamic variable**

```
shared_ptr<int> ptr_a = make_shared<int>(10);
```


References

- Lippman Chapter 12
- Deb Haldar. Top 10 dumb mistakes to avoid with C++ 11 smart pointers. http://www.acodersjourney.com/2016/05/top-10-dumb-mistakes-avoid-c-11-smart-pointers/

Credits

- Jason Rassi. Introduction to C++ smart pointers. http://cs.brown.edu/~jrassi/nyc_cpp_meetup_20131107_smart_pointers.pdf

Readings

Don't Use get to Initialize or Assign Another Smart Pointer!

```
shared_ptr<int> p(new int(42)); // reference count is 1
int *q = p.get(); // ok: but don't use q in any way that might
                  // delete its pointer

{ // new block
    // undefined: two independent shared_ptrs point to the same memory
    shared_ptr<int> p2(q);
} // block ends, q is destroyed, and the memory to which q points is freed

int foo = *p; // undefined; the memory to which p points was freed
```